Multi-Splat Representations

Segmentation and Object Manipulation in Gaussian Splatting

Bavo Verstraeten

June 1, 2025

Abstract

Gaussian splatting is a novel technique for real-time rendering of complex scenes. This thesis improves the usability and accuracy of Gaussian splats in the Unity engine by extending an existing open-source Unity extension.

Two key challenges are addressed. First, the original Unity extension performs per-GameObject sorting and rendering of splats, leading to visual artifacts when multiple splat objects overlap. To resolve this, the existing rendering pipeline was adapted to perform global per-splat sorting, enabling correct depth ordering across all splats in the scene.

Second, the preparation of splats for animation in Unity traditionally requires manual and error-prone segmentation. To facilitate this process, a semi-automatic segmentation pipeline was developed by combining the tools SAM, SAM2, and SAGD. This approach automates much of the segmentation process, while retaining user control for refinement and correction, ultimately producing hierarchical GameObjects from splat data.

These contributions enhance both rendering fidelity and workflow usability, advancing the integration of Gaussian splatting into real-time interactive applications.

Contents

1 Introduction

Recent advances in 3D scene representation have shifted from traditional mesh-based models to point-based approaches. Among these, Gaussian splatting (Kerbl et al. 2023) has emerged as a promising technique for real-time rendering of photorealistic scenes, leveraging a collection of 3D Gaussians to represent surfaces without relying on explicit geometry. This method achieves high visual fidelity while maintaining interactive frame rates, making it an attractive alternative to neural radiance fields (NeRF) and triangle meshes for real-time applications.

This thesis began not with a formal hypothesis or a narrowly defined research question, but with a practical goal: to explore what it would take to animate a Gaussian Splat inside Unity, even if through improvised or non-standard means, to enable their use in interactive applications. The initial objective was to implement a functioning system, and to let the problems encountered along the way guide the direction of the project.

This hands-on approach quickly revealed limitations in the widely referenced Unity Gaussian Splatting extension (Pranckevičius 2023) that adopts a per-object rendering pipeline. In this approach, each splat object is sorted and rendered independently, leading to incorrect visual results when multiple splat objects overlap in a scene.

With the rendering issue addressed, attention shifted to another practical barrier: the manual effort involved in preparing splat data for animation, particularly the segmentation of splats into hierarchical GameObjects. Recognizing that this bottleneck could hinder creative workflows, the project expanded to include the development of a semi-automatic segmentation pipeline aimed at balancing automation with user control.

The result of this process is a set of improvements that emerged directly from a workflow-oriented, problem-driven journey that reflects the realities of adapting cutting-edge graphics techniques to real-time, interactive settings.

The structure of this thesis reflects that same exploratory trajectory. It begins with a background chapter, introducing Gaussian splatting and its origin, followed by an explanation of the existing Unity extension's rendering pipeline. From there, each subsequent chapter follows the chronological order in which challenges arose and solutions were developed. First, the rendering issue and its resolution are discussed; next, the focus turns to the segmentation workflow and its implementation. This structure mirrors the iterative and hands-on approach of the project, allowing the reader to follow the evolving problem space and its solutions step by step.

2 Background

2.1 From Meshes to Gaussian Splatting

2.1.1 Triangle Meshes

In the field of computer graphics, the efficient rendering of complex 3D scenes has consistently posed a significant challenge. Traditional rendering techniques predominantly rely on triangle meshes (see Figure 1), where the representation of surfaces is approximated through a network of interconnected triangles. While these methods have proven effective in many applications, they encounter limitations when dealing with highly detailed or non-uniform data. Achieving photorealistic results in such cases necessitates considerable memory and computational resources. Furthermore, these approaches are heavily dependent on manually crafted 3D models, a process that is both time-consuming and requires a skilled artist. The generation of models from

video or a collection of images offers the potential for a significantly simpler and more scalable solution.

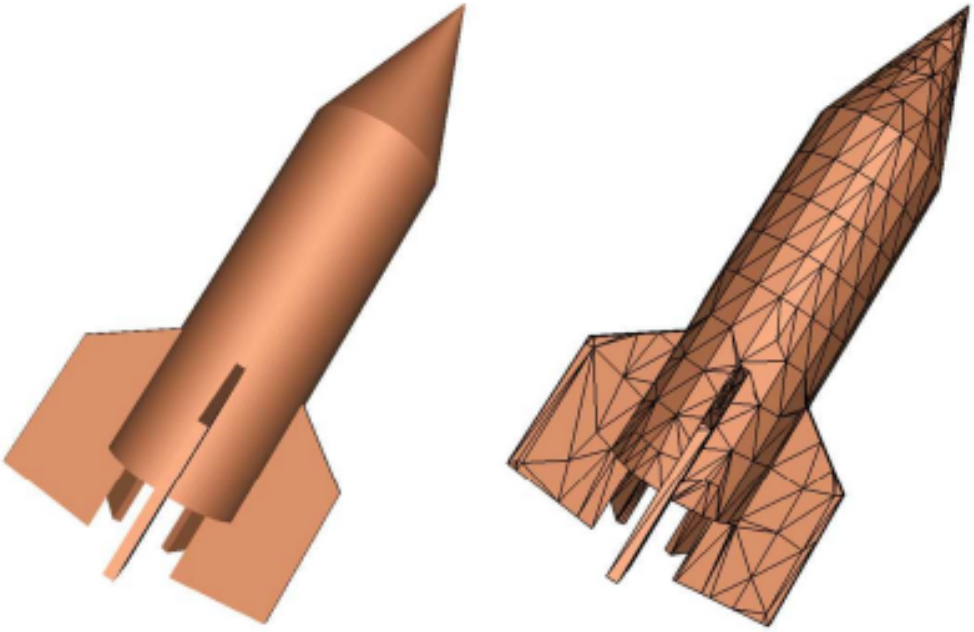

Figure 1: Example of a triangle mesh, the most commonly used method for representing 3D objects in computer graphics. Image is taken from Schaefer and Warren 2004.

2.1.2 Neural Radiance Fields

One popular approach to addressing these challenges is Neural Radiance Fields (NeRF) (Mildenhall et al. 2020). This method employs a neural network to learn a mapping from 3D position and viewing direction to a density and an RGB color. The network is trained on a video or a collection of images, allowing it to reconstruct a scene by learning the underlying volumetric representation. During rendering, a ray is cast through each pixel of the screen, and multiple points along the ray are sampled. The neural network takes the position of each point and the viewing direction of the ray as input, producing color and density values. These outputs are then aggregated to compute the final color of the pixel, enabling the generation of photorealistic scenes without the need for manually crafted 3D models (see Figure 2). However, due to the necessity of querying the neural network multiple times per pixel, this approach is computationally expensive and not suitable for real-time rendering.

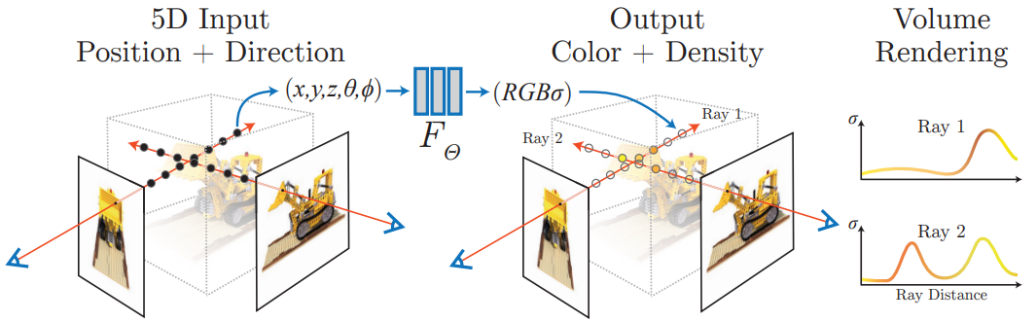

Figure 2: Rendering a scene using Neural Radiance Fields. Image is taken from Mildenhall et al. 2020.

2.1.3 Gaussian Splatting

In 2023, a method was introduced that retained the advantages of Neural Radiance Fields, while achieving real-time processing: Gaussian Splatting (Kerbl et al. 2023). Like Neural Radiance Fields, Gaussian Splatting uses

a video or a collection of images as input to automatically generate the 3D scene data. However, rather than relying on a neural network to represent the data, the scene is represented as a collection of numerous small 3D Gaussians (see Figure 3).

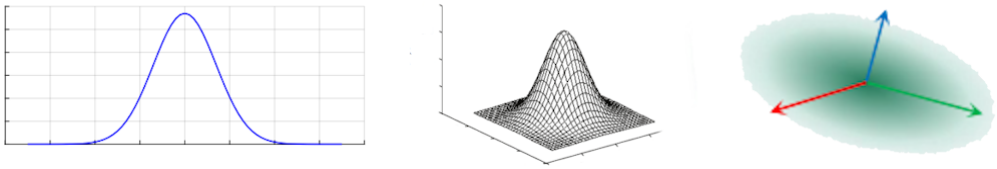

Figure 3: Illustration of Gaussian functions in different dimensions, shown left to right: 1D Gaussian (T. T. Nguyen, P. Nguyen, and Bouchara 2023), 2D Gaussian (Sathyamoorthy and Palanikumar 2006), and 3D Gaussian (Jiang et al. 2023).

Each Gaussian is defined by several attributes that determine its appearance and behavior during rendering:

- 3D coordinates
- Opacity (transparency)
- Anisotropic covariance: The degree to which the Gaussian is 'stretched' along each of the three axes independently
- Spherical Harmonics: The color of the Gaussian, which varies depending on the viewing angle

The key advantage of this approach is the way the scene is rendered. Instead of relying on rays or neural networks, which are computationally expensive, Gaussian Splatting leverages a rasterization technique. In this process, the 3D Gaussians are projected directly onto the 2D screen space, where they are rendered as small, disk-like splats. These splats are then blended together based on their spatial overlap, opacity, and color attributes.
Furthermore, state-of-the-art triangle mesh rendering also relies on rasterization, meaning that this approach already benefits from many well-established optimizations in graphics hardware. These optimizations, including efficient handling of spatial data and parallel processing, are directly applicable to Gaussian Splatting. By utilizing the same rendering pipeline, Gaussian Splatting can leverage these optimizations for further efficiency gains, making it an ideal choice for real-time rendering applications.

2.2 Unity Extension

2.2.1 Introduction

When adapting this technique to interactive applications —such as games—several engines come to mind, including Unity, Godot, and Unreal Engine. Among

these, Unity was chosen due to prior experience with the engine. Additionally, a widely referenced GitHub project (Pranckevičius 2023) implementing Gaussian Splatting in Unity provides a strong foundation for further research. This project stands out not only for its recognition within the community (Saucier 2024; Rubloff 2024), but also for its developer, Aras Pranckevičius, a former Unity engineer with 15 years of experience working on the engine. His expertise suggests that the rendering pipeline adaptations are both well-optimized and efficiently integrated, making this implementation a suitable choice for exploration and development.

Before examining the objectives of this thesis and the adaptations made to the selected implementation, it is essential to first outline the relevant aspects of its rendering pipeline. The discussion will focus solely on components that directly influence the rendering process. Certain aspects, such as the compression and storage of Gaussian Splats in files, among others, fall outside the scope of this explanation and will not be covered.

2.2.2 Adding a Gaussian Splat to the Scene

The selected project does not implement the training of Gaussian Splats but instead relies on pretrained models obtained from an external source. These models are not used in their original format; rather, the project includes a Unity Editor extension that converts them into a more efficient and compressed representation (see Figure 4). As previously mentioned, the specifics of this compression process have minimal impact on rendering and will therefore not be discussed in detail. Once the compressed asset is generated, the user must manually create a new empty GameObject and attach the `GaussianSplatRenderer` script to it. This script includes a field labeled Asset, into which the compressed asset must be assigned. Upon doing so, the Gaussian Splat appears in the scene. Like any GameObject in Unity, it can be translated, rotated, and scaled. This process can be repeated to

instantiate multiple Gaussian Splats within the same scene.

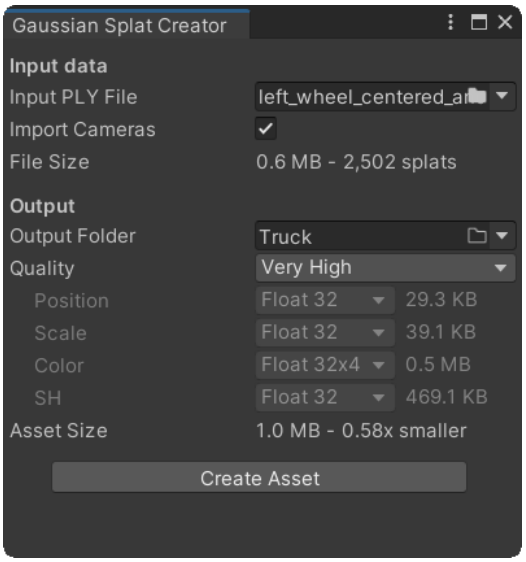

Figure 4: The Unity Editor extension included in the project for converting PLY files into the required compressed format.

2.2.3 Initializing the Rendering Process: The Role of `OnEnable`

When no GameObjects with an attached `GaussianSplatRenderer` script are present, the renderer follows the standard rendering pipeline. However, once such a GameObject is enabled, its `OnEnable` method is executed, performing several initialization tasks. The method is outlined in the following code fragment:

Code 1: `onEnable` of `GaussianSplatRenderer`

```
public void OnEnable()
{
  // Check if necessary shaders and compute shaders are
      available

  // Initialize materials (details omitted, see later)
```

```
  // Initialize the sorter

  GaussianSplatRenderSystem.instance.RegisterSplat(this);

  // Initialize all the graphics buffers
}
```

As demonstrated in the code segment, the `OnEnable` method calls the `RegisterSplat` function of `GaussianSplatRenderSystem`, a singleton class. This class plays a pivotal role in managing the multiple `GaussianSplatRenderer` objects within the scene and is responsible for executing the changes that modify the rendering process. The implementation of the `RegisterSplat` function is as follows

Code 2: `RegisterSplat` of `GaussianSplatRenderSystem`

```
public void RegisterSplat(GaussianSplatRenderer r)
{
  if (m_Splats.Count == 0)
  {
    if (GraphicsSettings.currentRenderPipeline == null)
    Camera.onPreCull += OnPreCullCamera;
  }
  m_Splats.Add(r, new MaterialPropertyBlock());
}
```

The `GaussianSplatRenderSystem` maintains a dictionary of all enabled `GaussianSplatRenderer` objects. When this function is invoked, and no other `GaussianSplatRenderer` are registered, the method subscribes to the `Camera.onPreCull` event, linking it to the `OnPreCullCamera` method.
When a `GaussianSplatRenderer` is disabled, it clears all associated buffers and materials, removes the object from the internal dictionary, and unsubscribes from the `onPreCull` event when the last `GaussianSplatRenderer` is removed, effectively managing resources and maintaining system efficiency.

2.2.4 The Rendering Pipeline

Each frame, the Unity engine performs culling before rendering the scene, determining which objects lie within the camera's view and should be processed for rendering. Since splats may appear outside of the camera's view, the moment just prior to culling is an optimal point at which to prepare these splats and enqueue their draw procedure in the command buffer for execution during the rendering stage. That moment is when `onPreCull` calls are made.
As previously mentioned, when the first `GaussianSplatRenderer` is enabled, the `OnPreCullCamera` function is linked to `onPreCull`. The implementation of this function is as follows:

Code 3: `OnPreCullCamera` of `GaussianSplatRenderSystem`

```
void OnPreCullCamera(Camera cam)
{
  if (!GatherSplatsForCamera(cam))
  return;

  InitialClearCmdBuffer(cam);

  m_CommandBuffer.GetTemporaryRT(
     GaussianSplatRenderer.Props.GaussianSplatRT, -1, -1, 0,
     FilterMode.Point, GraphicsFormat.R16G16B16A16_SFloat);
  m_CommandBuffer.SetRenderTarget(
     GaussianSplatRenderer.Props.GaussianSplatRT,
     BuiltinRenderTextureType.CurrentActive);
  m_CommandBuffer.ClearRenderTarget(RTClearFlags.Color, new
     Color(0, 0, 0, 0), 0, 0);

  // add sorting, view calc and drawing commands for each
     splat object
  Material matComposite = SortAndRenderSplats(cam,
     m_CommandBuffer);
```

```
    // compose
    m_CommandBuffer.BeginSample(s_ProfCompose);
    m_CommandBuffer.SetRenderTarget(
        BuiltinRenderTextureType.CameraTarget);
    m_CommandBuffer.DrawProcedural(Matrix4x4.identity,
        matComposite, 0, MeshTopology.Triangles, 3, 1);
    m_CommandBuffer.EndSample(s_ProfCompose);
    m_CommandBuffer.ReleaseTemporaryRT(
        GaussianSplatRenderer.Props.GaussianSplatRT);
}
```

This function performs 5 key operations:

- `GatherSplatsForCamera`: This function first iterates over all key-value pairs in `m_Splats` and selects those that have a valid compressed asset assigned, storing them in the list `m_ActiveSplats`. Subsequently, this list is sorted based on the depth of each entry's transform in the camera's coordinate space. The implementation of this function will be presented later.

- `InitialClearCmdBuffer`: This function initializes and clears a command buffer, `m_CommandBuffer`, before attaching it to the camera. By doing so, draw commands, which are created in later parts of the code, can be enqueued into this buffer. Each draw command ensures execution during the later rendering process if it is not culled.

- The function creates a temporary render target, `GaussianSplatRT`, sets it as the active render target, and clears it. Consequently, any draw calls issued before the render target is switched again will render to this temporary target rather than directly to the camera's output.

- `SortAndRenderSplats`: This function is responsible for the main calculations. It processes all collected splats and enqueues the necessary draw commands to render them into `GaussianSplatRT`. The implementation of this function will be presented later.

- The render target is reset to the camera's primary output, after which a `DrawProcedural` call is issued using a designated material. This material is associated with a shader that generates a single triangle that covers the entire screen in its vertex shader. The fragment shader subsequently reads data from `GaussianSplatRT` and applies gamma correction. This `DrawProcedural` call enqueues the rendering of this full-screen triangle, ensuring that the Gaussian Splats are correctly displayed in the final rendered image.

To fully understand what gets rendered to the screen, it is essential to take a close look at `SortAndRenderSplats`:

Code 4: `SortAndRenderSplats` of `GaussianSplatRenderSystem`

```
public Material SortAndRenderSplats(Camera cam,
    CommandBuffer cmb)
{
  Material matComposite = null;
  foreach (var kvp in m_ActiveSplats)
  {
    var gs = kvp.Item1;
    matComposite = gs.m_MatComposite;
    var mpb = kvp.Item2;

    // sort
    var matrix = gs.transform.localToWorldMatrix;
    if (gs.m_FrameCounter % gs.m_SortNthFrame == 0)
    gs.SortPoints(cmb, cam, matrix);
    ++gs.m_FrameCounter;

    kvp.Item2.Clear();

    // [Omitted: Set some variables related to debugging]

    gs.SetAssetDataOnMaterial(mpb);
    mpb.SetBuffer(GaussianSplatRenderer.Props.SplatChunks,
       gs.m_GpuChunks);
```

```
    mpb.SetBuffer(GaussianSplatRenderer.Props.SplatViewData,
        gs.m_GpuView);

    mpb.SetBuffer(GaussianSplatRenderer.Props.OrderBuffer,
        gs.m_GpuSortKeys);
    mpb.SetFloat(GaussianSplatRenderer.Props.SplatScale,
        gs.m_SplatScale);
    mpb.SetFloat(GaussianSplatRenderer.Props.SplatOpacityScale,
        gs.m_OpacityScale);
    mpb.SetFloat(GaussianSplatRenderer.Props.SplatSize,
        gs.m_PointDisplaySize);
    mpb.SetInteger(GaussianSplatRenderer.Props.SHOrder,
        gs.m_SHOrder);
    mpb.SetInteger(GaussianSplatRenderer.Props.SHOnly,
        gs.m_SHOnly ? 1 : 0);
    mpb.SetInteger(GaussianSplatRenderer.Props.DisplayIndex,
        gs.m_RenderMode ==
        GaussianSplatRenderer.RenderMode.DebugPointIndices ?
        1 : 0);
    mpb.SetInteger(GaussianSplatRenderer.Props.DisplayChunks,
        gs.m_RenderMode ==
        GaussianSplatRenderer.RenderMode.DebugChunkBounds ? 1
        : 0);

    cmb.BeginSample(s_ProfCalcView);
    gs.CalcViewData(cmb, cam, matrix);
    cmb.EndSample(s_ProfCalcView);

    // [Omitted: Set some variables related to debugging]

    cmb.BeginSample(s_ProfDraw);
    cmb.DrawProcedural(gs.m_GpuIndexBuffer, matrix,
        displayMat, 0, topology, indexCount, instanceCount,
        mpb);
    cmb.EndSample(s_ProfDraw);
}
```

```
  return matComposite;
}
```

This code block is extensive and encompasses multiple operations. To facilitate a clearer understanding, it will be analyzed step by step, broken down into its fundamental components. Additionally, certain sections of the code are dedicated to debugging or timing purposes, which are not essential for understanding the main rendering pipeline.

The function iterates over all `GaussianSplatRenderer` objects stored and ordered in the `m_ActiveSplats` list by `GatherSplatsForCamera`. It is important to emphasize that within this function, all subsequent operations are performed on a single `GaussianSplatRenderer` object at a time, as each iteration of the loop processes one individual `GaussianSplatRenderer`.

At regular intervals, specifically every set number of frames, the splats undergo sorting. This operation is managed by a custom sorter class, which was initialized during the `OnEnable` method. The class implements a specialized sorting algorithm known as "Device Radix Sort," leveraging the full potential of the GPU to ensure optimal performance. According to the comments in that class, this algorithm was adapted from another GitHub project (Smith 2024). As with the sorting of the `GaussianSplatRenderer` objects, the depth of each splat relative to the camera's coordinate space is used as the key for sorting.

Similar to the final `DrawProcedural` invocation of the previous function, a material shader is employed at the end of each iteration of the loop to render the splats. Many of the operations within this loop are dedicated to populating graphics buffers with the necessary data for these shader computations. The most significant step in this process is the invocation of `CalcViewData`, which dispatches splat-related information to a compute

shader. This compute shader determines the screen-space position of each splat using the object's transform matrix, the `cam.worldToCameraMatrix`, and the `cam.projectionMatrix`. Additionally, it calculates its independent horizontal and vertical extents, and derives its color and transparency based on the viewing angle and spherical harmonics. The computed results are then stored in a graphics buffer for subsequent use.

Finally, `DrawProcedural` is called, enqueueing the rendering of this data for execution later in the rendering pipeline. As before, this rendering is performed using a material shader, a different one from the material shader used in `OnPreCullCamera`. Each splat is represented as two triangles, with the vertex shader transforming the computed screen position and stretch extents into appropriate triangle coordinates.

As a brief reminder, as stated earlier, this `DrawProcedural` invocation does not render directly to the camera's output but instead to the temporary render target `GaussianSplatRT`. Additionally, due to the sorting performed at the beginning of the loop, splats farther from the camera are rendered before those closer to it, ensuring correct per-object rendering order.

With these steps described, the overall structure of the rendering pipeline for the Gaussian Splats becomes clear. The coordination between sorting, compute shaders, and material shaders ensures that the splats are accurately positioned, stretched, and rendered according to the camera's view.

3 Part 1: Correcting the Rendering Order

3.1 Introduction

At the outset of this research, no specific focus or predefined problem statement had been established. The initial objective was exploratory in nature:

to experiment with the existing Unity extension and investigate the process of importing and animating a Gaussian Splat within the engine.
Throughout this exploratory process, particular attention was given to identifying tasks that were difficult to perform, as well as outputs that appeared incorrect or inconsistent. These observations served as the basis for defining the core focus areas of the thesis.

3.2 Preparing the scene

To initiate this phase of the research, it was necessary to select a suitable pre-trained Gaussian Splat model. Ideally, the chosen model would consist of multiple visually distinct segments, making it intuitive and practical for experimentation with segmentation and animation. The primary models considered were those released by the authors of the original Gaussian Splatting paper. Among them, the Kitchen scene includes a small LEGO bulldozer—an object that is compact, easily recognizable, and composed of several clearly separable parts (see Figure 5). These characteristics made it an ideal candidate for the intended use case.

Figure 5: The Kitchen scene from the official Gaussian Splatting dataset, featuring a small LEGO bulldozer. This object was selected for its compact structure and clearly defined components, making it well-suited for segmentation and animation experiments.

Obviously, not the entire scene is necessary for these purposes; only the LEGO bulldozer is required. Fortunately, the Unity extension being used includes a tool designed specifically for this type of task: cutouts. Within the GaussianSplatRenderer component of the relevant GameObject in the Unity Editor, a cutout can be added to the Gaussian Splat. This tool creates a cutout GameObject (either a box or an ellipsoid) as a child of the selected object. The cutout can be translated, rotated, and scaled like any other GameObject. Any splats of the selected object outside of the cutout are excluded from rendering. Additionally, there is an invert option that reverses the effect, making only the splats within the cutout invisible. A button allows the Gaussian Splat to be saved as a PLY file, accounting for all cutout modifications. This functionality was used to generate a PLY file and associated asset containing only the bulldozer, omitting the rest of the scene (see Figure 6).

Figure 6: The LEGO bulldozer extracted from the Kitchen scene using the cutout tool of the Unity extension for Gaussian splatting. This tool isolates the bulldozer by excluding all other parts of the scene.

3.3 Animating a segment

To animate the LEGO bulldozer, the next step was to segment it into multiple parts, with each part represented as a separate GameObject and parented to the main bulldozer GameObject. Instead of segmenting the entire model at once, segmentation began with a single part to test the process. The first part selected for segmentation was one of the wheels at the back of the bulldozer. Using the cutout tool, a cutout was created that perfectly covered the wheel, allowing for the generation of a PLY file containing only the wheel. To isolate the bulldozer without the wheel, the same cutout was applied with the invert option enabled. However, when the separate wheel GameObject was rendered, it appeared behind the rest of the bulldozer, resulting in an unexpected rendering error (see Figure 7).

Figure 7: A rendering error when loading the separate wheel-GameObject.

Fortunately, the error can be resolved by setting the z-value of the wheel GameObject's position to 0.001. Nonetheless, the need for this workaround already indicates that something is fundamentally wrong with the rendering process.

The next objective was to rotate the wheel around its own axis. However, this revealed a new challenge: the center of the wheel's GameObject was inherited from the original full scene and therefore located far from the actual center of the wheel. In addition, the existing rotation axes were misaligned with the intended axes of rotation (see Figure 8).

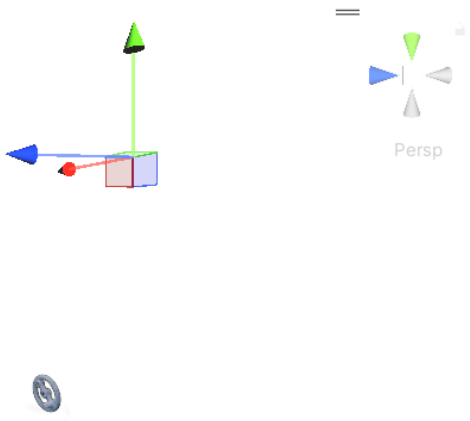

Figure 8: The center of the wheel's GameObject is positioned far from the geometric center of the wheel, with misaligned axes of rotation. This misalignment complicates direct rotation around the wheel's local axis.

A seemingly straightforward solution is implemented to achieve what appears to be the desired result. A small, distinctively colored cube is created and designated as the parent of the wheel GameObject. The wheel is then repositioned and rotated relative to the cube so that its center aligns with the center of the cube and its plane is parallel to the cube's axes (see Figure 9). Once this alignment is achieved, the cube is made invisible. By rotating the parent, the wheel now rotates correctly around the intended axis. The same procedure was applied to the base of the bulldozer to achieve similar control.

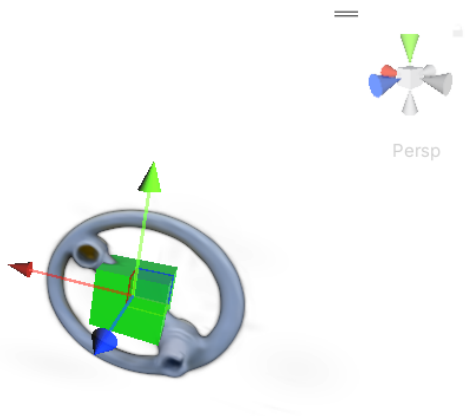

Figure 9: A cube GameObject is assigned as the parent of the wheel GameObject. The wheel is repositioned and rotated so that its center and axes align with those of the cube. This configuration enables rotation around the intended axis.

A simple animation was created using Unity's built-in Animation feature to rotate the wheel around its axis. However, this exposed a flaw in the proposed solution: the wheel exhibited flickering during rotation, appearing in front of the bulldozer at certain angles and behind it at others (see Figure 10). Further inspection revealed the underlying cause: rotating the wheel around the new axis also rotates the center of the Gaussian Splat itself—still located at the center of the original full scene—around that axis. Since the per-GameObject renderer relies on this center point to determine the rendering order based on its depth to the camera, any displacement of the center could lead to a different rendering order. Consequently, an alternative solution was required to prevent the center from moving during rotation.

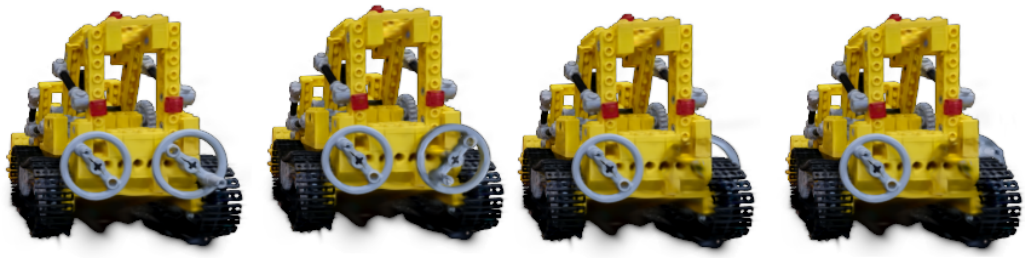

Figure 10: Visualization of the rendering issue caused by rotation around its axis. The same wheel is shown at 0°, 90°, 180°, and 270° of rotation. Depending on the rotation angle, the wheel is rendered either behind or in front of the bulldozer due to improper depth sorting based on the rotated center of the Gaussian Splat.

The only way to eliminate this visual artifact is to align the origin of the Gaussian Splat with the center of the wheel. This requires modifying the underlying splat data in the corresponding .ply file. To achieve this, a small Python script was developed to automatically adjust the position values of each splat.

Initially, it seemed straightforward to have the script infer the desired center point automatically. Two potential approaches were implemented: calculating the average of the position values of all splats, and calculating the midpoint between the minimum and maximum position values along each axis. However, testing revealed that both methods resulted in a center point that was slightly offset, leading to imperfect alignment (see Figure 11). While other automatic methods, such as taking the size of the splats into account, could have been explored, they were not pursued. Instead, the decision was made to allow the user to manually input an offset value for greater control, particularly in cases where complex and irregular shapes may have a "desired" center that is difficult to compute automatically. The desired offset can be determined by applying the earlier alignment procedure with the cube and copying the position values of the wheel GameObject relative to its parent. The Python function solely modifies the center point of the GameObject,

leaving the alignment of the axes unchanged, which is still achieved through the cube method. The most relevant part of the resulting Python script is provided below. The plyfile library (Ranjan 2021) was used to load, edit and store the splat data:

Code 5: Three potential functions for recentering the Gaussian Splat. The desired function is called on each of the three axes.

```
def recenter_axis_average(ply_content, axis):
  axis_poses = ply_content.elements[0].data[axis]
  axis_avg = axis_poses.mean()
  axis_poses -= axis_avg

def recenter_axis_box(ply_content, axis):
  axis_poses = ply_content.elements[0].data[axis]
  axis_middle = (axis_poses.max()+axis_poses.min())/2
  axis_poses -= axis_middle

def recenter_axis_offset(ply_content, axis, offset):
  axis_poses = ply_content.elements[0].data[axis]
  axis_poses += offset
```

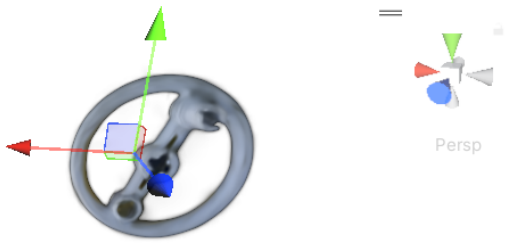

Figure 11: Automatically recentering based on the average of the position values results in a slight offset. Recentering using the midpoint of the bounding box yields a similar misalignment.

3.4 Moving the camera

The wheel now both renders and rotates as intended, but these results were observed under an unambiguous camera perspective. Once the camera was moved or rotated, it became apparent that the rendering order had another issue. Specifically, rotating the camera altered the rendering order, while translating the camera had no effect. This behavior is the opposite of what one would expect for correct depth-based rendering. To better illustrate this issue—and to further validate the solution from the previous step—a second wheel was segmented into its own GameObject, following the same procedure described earlier (see Figure 12).

Figure 12: Demonstration of incorrect rendering order under camera transformation. Each column corresponds to a fixed camera rotation, while each row corresponds to a fixed camera position. The observed rendering order remains the same across rows within each column, indicating that the order is determined solely by camera rotation. Correct behavior would require the rendering order to change across rows (i.e., with camera position).

It becomes evident that the problem lies within the renderer itself; therefore, adjusting the GameObjects or the splats alone cannot provide a solution. To identify the root cause, a closer examination of the Gaussian Splat rendering process is necessary. A general overview of the pipeline was provided earlier in this thesis. In summary:
The renderer maintains a data structure that holds all enabled Gaussian

Splats. First, it sorts these GameObjects based on the depth of each entry's transform in the camera's coordinate space. Then, for each GameObject, the individual splats are sorted, and the screen position and shape of each are computed and rendered.
The flaw is immediately apparent in this summary: the method of sorting the GameObjects.
When the camera is translated (without rotation), the relative depth of each GameObject remains unchanged. That is, if the depth of one GameObject increases by a certain amount, all others increase by the same amount, leaving the rendering order unaffected.
Conversely, when the camera is rotated, the depth of each GameObject relative to the camera's plane changes differently, resulting in a new rendering order.
The code of this functionality can be found below:

Code 6: The sort key used to determine the rendering order

```
var trA = a.Item1.transform;
var trB = b.Item1.transform;
var posA = camTr.InverseTransformPoint(trA.position);
var posB = camTr.InverseTransformPoint(trB.position);
return posA.z.CompareTo(posB.z);
```

To resolve this, a different metric is required for sorting the GameObjects. Using the Euclidean distance to the camera itself satisfies the requirements. When the camera is translated, each GameObject's distance to the camera changes independently, ensuring the rendering order updates correctly. When the camera is rotated, the distances remain unchanged, so the rendering order is preserved. This new sorting method thus addresses the core issue.
It is also worth noting that only the relative distances between GameObjects matter for determining the correct rendering order. This means that computing the exact Euclidean distance—including taking the square root of the squared distance—is not required. Instead, the squared distance can

be used directly for sorting, as it preserves the same relative ordering. This approach avoids unnecessary square root calculations, improving efficiency without affecting correctness. The new code can be found below:

Code 7: The newly implemented sort key to determine the rendering order

```
var trA = a.Item1.transform;
var trB = b.Item1.transform;
Vector3 APos = trA.position;
Vector3 BPos = trB.position;
float sqrDistanceA = (camPos - APos).sqrMagnitude;
float sqrDistanceB = (camPos - BPos).sqrMagnitude;
return sqrDistanceA.CompareTo(sqrDistanceB);
```

The new behavior is significantly closer to the expected result. In most camera positions and rotations, the body and wheels now render in the correct order. However, certain edge cases remain where the rendering is still incorrect (see Figure 13).

Figure 13: When the wheel should be rendered behind the bulldozer, but the camera is positioned near the threshold where the rendering order switches, the wheel is incorrectly rendered in front of the bulldozer. The left and right images have identical camera rotation, with only a minimal difference in camera position.

The core of the problem is that the renderer compares only the center points of each GameObject to determine depth order. In this scenario, the center of the bulldozer is located far from the region where overlap with the wheel actually occurs. While this comparison yields correct results from many

viewing positions, it fails when the camera is positioned near or between the two center points, where the relative depth becomes ambiguous or misleading. For nested GameObjects, there is typically a logical connection point—namely, the position where the child segment physically attaches to the parent. Ideally, the depth comparison would be made between this connection point and the center of the child segment (the wheel, in this case). However, accurately determining the exact location of this connection point is non-trivial. A practical alternative achieves a similar effect: introducing a pseudo-connection point along the line connecting the centers of the two GameObjects. This point is placed much closer to the center of the child than to that of the parent. By comparing this pseudo-connection point to the center of the child, a more stable and intuitive depth ordering can be achieved. This approach provides a better representation of the true spatial relationship between the two objects and ensures that the camera can no longer be positioned between the child's center and the new pseudo-connection point, thereby eliminating the ambiguous scenarios that previously caused incorrect rendering order. The code is provided below. Note that the method for determining the parent-child relationship is structured in this particular way due to the earlier alignment procedure, where each Gaussian Splat was made a child of an invisible cube to align the rotation axes.

Code 8: The final implemention of the sort key

```
Vector3 APos = trA.position;
Vector3 BPos = trB.position;
float ratio = 0.999f;
if (trA.parent == trB.parent.parent)
{
  APos = APos * (1 - ratio) + BPos * ratio;
}else if (trB.parent == trA.parent.parent)
{
  BPos = BPos * (1 - ratio) + APos * ratio;
}
```

```
float sqrDistanceA = (camPos - APos).sqrMagnitude;
float sqrDistanceB = (camPos - BPos).sqrMagnitude;

return sqrDistanceA.CompareTo(sqrDistanceB);
```

Finally, the wheels render as expected, as illustrated in Figure 13.

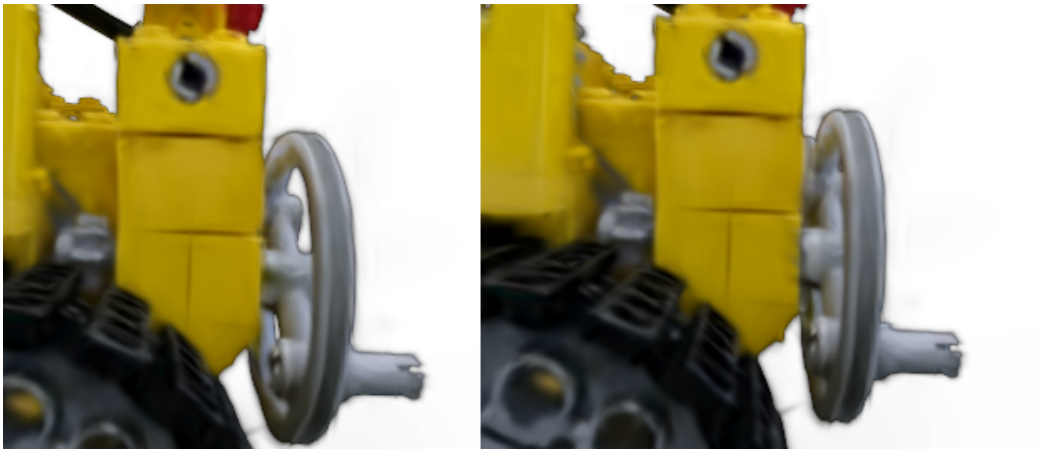

Figure 14: The camera is positioned at the critical point where the wheel should transition from being rendered behind the bulldozer to being rendered in front of it. The left and right images have identical camera rotations, with only a minimal change in camera position. As shown, the wheel now renders correctly across this boundary case.

3.5 Global per-splat sorting

With the basic subsegments now rendering correctly, the question arose whether to continue pursuing deeper improvements in this area or to shift focus toward other stages of the animation process that might require further research. It is important to recall that the overarching aim of this thesis is to identify and address the most significant challenges or errors across the entire pipeline.

To evaluate whether the current rendering solution was adequate enough to

justify moving on, more complex subsegments were isolated from the bulldozer model. In particular, the arm and its claw were selected for testing. The arm presents a notably more challenging case, as it wraps around the bulldozer, resulting in certain parts that must be rendered in front of the bulldozer, while others need to appear behind it (see Figure 15).

Figure 15: Regardless of the camera position, a visual artifact always occurs. When the arm is rendered behind the bulldozer (left), significant portions of its joints are lost. Conversely, when the bulldozer is rendered behind the arm, its front section is improperly occluded (right). The two images were captured with identical camera rotations and only a small difference in camera position.

After careful evaluation, it was concluded that the visual artifacts introduced by the current rendering approach are too disruptive to be ignored. This is particularly evident in interactive environments, such as games, where objects are frequently in motion and subject to continuous rotation. In such contexts, the visual inconsistencies become highly noticeable, making it worthwhile to invest further effort into resolving the issue.

Up to this point, most of the proposed solutions have relied on workaround-based approaches—methods that attempt to circumvent the core issue rather than address it directly. While such strategies have proven effective for simpler configurations, they fall short when applied to more complex geometries.

The fundamental limitation lies in the renderer's architecture: it operates on a per-GameObject basis, where entire GameObjects are first sorted relative to the camera, and only then are the splats within each GameObject sorted and rendered. This structure inherently fails in cases where different parts of a child object must appear both in front of and behind its parent. No amount of adjustment or approximation can resolve this limitation cleanly. A proper solution therefore requires a shift from per-GameObject sorting to fully global per-splat sorting—ensuring that every splat is sorted and rendered based solely on its position in space, regardless of GameObject hierarchy.

Since the core rendering system is now being modified directly, it is essential to first understand how the existing renderer functions. A detailed explanation of the current rendering pipeline is provided in Section 2.2.

To clearly present the implemented solution, it is helpful to begin by outlining the desired end result. The current rendering pipeline operates on a per-GameObject basis due to its underlying architecture. Each GameObject is processed independently, with its splat data prepared and stored in a separate graphics buffer. This buffer is generated by a compute shader dispatched specifically for that GameObject, since the shader requires access to the GameObject's local data and transform in order to populate the buffer correctly. Once populated, the buffer is passed to a `DrawProcedural` call to handle sorting and rendering of the splats contained within.
Ideally, the renderer would instead issue a single `DrawProcedural` call that processes a unified graphics buffer containing the data for all Gaussian splats in the scene. This would enable sorting and rendering on a per-splat level, thereby resolving the compositing issues observed with overlapping objects. However, constructing such a unified buffer in an efficient manner—without incurring significant runtime overhead or increased memory usage—is a non-

trivial challenge.

The proposed solution is to employ a large, global graphics buffer that consolidates the processed splat data for all objects. Rather than merging the individual per-GameObject buffers into this global buffer at the end of each computation—a process that would introduce additional complexity and runtime overhead—the system is designed so that all compute shaders write directly into the shared buffer. Specifically, the buffer stores the computed screen positions, stretching factors, and color data for the splats, just as was done in the per-GameObject buffers, but now combined into a single structure. To facilitate this, the shader must be informed of the specific location within the global buffer where its output should be written. This is accomplished by associating each `GaussianSplatRenderer` instance with an offset value that specifies its starting position in the buffer. Conveniently, a dictionary that maps `GaussianSplatRenderer` instances to metadata already exists, making it an ideal structure for storing these offsets. The size of the global buffer is dynamically determined and set to exactly match the total number of splats across all enabled `GaussianSplatRenderer` instances

To implement this global system, much of the logic previously contained within `GaussianSplatRenderer` is transferred to the `GaussianSplatRenderSystem`, which now manages the shared global buffer. Each time a `GaussianSplatRenderer` is enabled or disabled, the system creates a new global buffer of the appropriate size, and the offsets are updated accordingly. When a new `GaussianSplatRenderer` is enabled, its offset is set to the size of the buffer prior to its addition, effectively placing its data at the end. When a `GaussianSplatRenderer` is disabled, the system accounts for the possibility that the removed splats are positioned anywhere within the buffer. To maintain a contiguous structure, the offsets of all `GaussianSplatRenderer` instances whose offsets are greater than

that of the disabled segment are decreased by the number of splats in the disabled segment. This ensures that the global buffer remains correctly partitioned at all times. The code is provided below. Note that this approach does increase complexity, but importantly, this complexity is incurred only when a `GaussianSplatRenderer` is enabled or disabled, rather than every frame. If such events occur frequently, the new pipeline could indeed introduce additional performance overhead. However, this trade-off is limited to the frequency of structural changes in the scene, and its impact should be considered in the context of the specific application's dynamics.

Code 9: New RegisterSplat and UnregisterSplat

```
public void RegisterSplat(GaussianSplatRenderer r)
{
  if (m_Splats.Count == 0)
  {
    m_FrameCounter = 0;
    if (m_CSSplatUtilities == null)
      m_CSSplatUtilities = r.m_CSSplatUtilities;
    if(m_Sorter == null)
      m_Sorter = new GpuSorting(m_CSSplatUtilities);
    if (GraphicsSettings.currentRenderPipeline == null)
      Camera.onPreCull += OnPreCullCamera;
  }

  m_Splats.Add(r, (new MaterialPropertyBlock(),
     totalSplatCount));

  totalSplatCount += r.asset.splatCount;

  m_GpuViewCombined?.Dispose();
  m_GpuViewCombined = null;
  m_GpuViewCombined = new
     GraphicsBuffer(GraphicsBuffer.Target.Structured,
     totalSplatCount, 40);
```

```
  // initialize sorter
}

public void UnregisterSplat(GaussianSplatRenderer r)
{
  int splatCount = r.asset.splatCount;

  if (!m_Splats.ContainsKey(r))
  return;

  int removedOffset = m_Splats[r].Item2;
  m_Splats.Remove(r);
  foreach (var key in m_Splats.Keys.ToList())
  {
    var value = m_Splats[key];
    if(value.Item2 > removedOffset)
    m_Splats[key] = (value.Item1,value.Item2 - splatCount);
  }
  totalSplatCount -= splatCount;

  m_GpuViewCombined?.Dispose();
  m_GpuViewCombined = null;

  if (m_Splats.Count == 0)
  {
    // clear camera buffers

    // clear global buffers

    Camera.onPreCull -= OnPreCullCamera;
  }
  else
  {
    m_GpuViewCombined = new
       GraphicsBuffer(GraphicsBuffer.Target.Structured,
       totalSplatCount, 40);
    InitSortBuffers();
```

```
    }
}
```

While all processed splat data is now consolidated in a global buffer, the sorting of splats must also be addressed. In the original implementation, sorting of splats within a `GaussianSplatRenderer` occurred prior to the calculation of screen positions, with the sort key computed via a simple, dedicated compute shader. However, in the revised system, it is not feasible to perform this sorting step before the calculation of screen positions, as the splat data only becomes available in the global buffer after these calculations are complete. Fortunately, as is often the case in perspective rendering, the screen-space data is stored in homogeneous coordinates, with the depth value encoded in the fourth component. This allows the sorting to be performed directly on the fully populated global buffer, using the depth value extracted from the existing data. As a result, sorting of the global buffer can be implemented without introducing additional complexity or requiring extra data specifically to compute the sort key.

After the complete implementation of this global sorting and rendering system, with all components integrated and functioning cohesively, a fully correct rendering order has been achieved, as demonstrated in Figure 16.

Figure 16: The final rendering result after implementing global per-splat sorting. Notably, both wheels at the back, the arm, and the handle are all represented as separate GameObjects, yet are rendered correctly.

After further testing, a final issue was discovered, which fortunately proved to be straightforward to resolve: switching the asset assigned to a `GaussianSplatRenderer` caused the editor to crash. This problem arose because changing the asset neither enabled nor disabled the associated GameObject, yet it still altered the number of splats represented by that object.
The solution is almost trivial. Each `GaussianSplatRenderer` already stores the hash of its assigned asset within the asset itself. By also storing this hash in a local variable within the `GaussianSplatRenderer`, a straightforward comparison can be performed during each update cycle to determine whether the stored hash matches the hash currently stored in the asset. If the hashes do not match, it indicates that the asset has been changed.
To resolve the resulting issues with the global buffer size and offset management, the simplest solution is to call `UnregisterSplat` followed by `RegisterSplat` for the affected `GaussianSplatRenderer`. Additionally, the number of splats associated with the current asset is stored in a local variable. This is essential because once the asset is changed, the previous splat count is no longer directly accessible, making it impossible to properly adjust the global buffer size and offsets during unregistration. To accommodate `UnregisterSplat` for this case, it can now take an optional `splatCount` argument, which by default uses the size of the current asset, as shown in the code below:

Code 10: A small adjustment to UnregisterSplat

```
public void UnregisterSplat(GaussianSplatRenderer r, int
    splatCount = -1)
{
  if(splatCount == -1){
    splatCount = r.asset.splatCount;
  }
  ...
}
```

3.6 Conclusion

The efforts detailed in this part of the thesis addressed critical rendering challenges in the Gaussian splatting pipeline. By transitioning from a per-GameObject to a fully global per-splat sorting and rendering system, the proposed solution resolved depth-ordering issues that previously resulted in visual artifacts and inconsistencies, particularly in complex scenes with overlapping splats. Through careful restructuring of the rendering architecture—including the introduction of a unified graphics buffer and adjustments to buffer management and sorting algorithms—the final implementation achieved correct and stable rendering behavior, even under dynamic camera movements and complex object hierarchies. These improvements not only enhance the visual fidelity of Gaussian splatting in Unity but also establish a robust foundation for further exploration, which will be the focus of the following section.

4 Part 2: Hierarchical Segmentation

4.1 Introduction

While the rendering pipeline has been significantly improved, it is evident that further refinements are possible. Certain features of the original extension, such as highlighting specific splats, are no longer immediately functional due to the shift from `GaussianSplatRenderer` to the centralized `GaussianSplatRenderSystem`. These issues, however, can be addressed by migrating the necessary functionality from the former to the latter.

As in the previous stage of this research, it was necessary to evaluate whether pursuing these refinements would be a valuable use of time, or if efforts would be better spent addressing new challenges. This time, the circumstances differ significantly: the corrected rendering system now operates reliably across

all tested scenes, with no critical issues affecting development or introducing performance bottlenecks.
Given that streamlining the code and addressing residual issues represents a considerable but straightforward task, it was decided to shift the focus of this thesis toward a more substantial and technically engaging problem.

To determine the next focus of this thesis, it was necessary to identify the most significant remaining challenge in the entire process—from starting with a large, complete scene to arriving at individual segments that can move and rotate correctly. With the rendering errors now resolved, there were no longer any immediate errors impeding further development. The next step was to assess which aspects of the process were most cumbersome or difficult and might therefore benefit from further improvement.

Upon careful reflection, it became evident that the primary bottleneck lies in a component that this thesis has thus far only briefly mentioned: the segmentation of the splats themselves.

As previously described, this process relies on so-called "cutouts". In summary, cutouts are defined regions—either cuboid or ellipsoidal—that exclude either all splats inside or all splats outside of them. Extracting precise details from complex objects often requires numerous small cutouts, resulting in a cumbersome and time-consuming workflow that must be repeated for each sub-object. Moreover, this method does not permit the splitting of individual splats. This limitation has a more significant impact than one might initially expect: the interiors of objects are not captured in training images, leading to chaotic and oversized splats. When attempting to isolate a sub-segment, particularly at connection points with neighboring segments, this often results in large splats either missing or protruding awkwardly, as they cannot be bisected using this method.

4.2 Relevant work

Gaussian splatting is a rapidly evolving and highly promising field, with numerous research papers being published each month across a wide range of specialized topics. These include areas such as automatic meshing (Guédon and Lepetit 2023), compression (Navaneet et al. 2024), physics (Feng et al. 2024), texturing (Chao et al. 2025), and many others. Unsurprisingly, segmentation has also emerged as a major area of focus within this landscape, with over 30 published papers already exploring various use cases and requirements.

The existing literature on segmentation can broadly be categorized into two groups. The first group focuses on the fundamental task of simple segmentation, aiming to extract objects or sub-objects from larger scenes. The second group extends this concept to semantic linking, where the goal is to associate each identified object with descriptive labels or to enable object generation based on textual prompts. This latter category is often referred to as "open-vocabulary detection."

While open-vocabulary detection techniques offer an interesting approach to user-friendly interaction, they are not the focus of this thesis. Although such methods could allow users to extract segments by describing them with natural language, incorporating these techniques would introduce a different direction for the work and require additional tools and methods. Moreover, open-vocabulary detection adds an extra layer on top of the core segmentation challenge, increasing overall complexity without directly addressing the underlying issue of precise geometric segmentation. For readers interested in exploring open-vocabulary detection methods, several representative works are available in the literature (Shi et al. 2023; Guo et al. 2024; Liang et al. 2024).

Another group of important papers, though not directly relevant to the objectives of this thesis, focuses on dynamic splats (Li et al. 2024; Ji et al. 2024). These approaches extend Gaussian splatting into the temporal domain, resulting in 4D splats where the fourth dimension represents time. Effectively, they model scenes as Gaussian splats evolving over video frames. However, since this thesis operates exclusively with static 3D splats, these dynamic approaches fall outside the scope of this work.

Finally, some papers focus on fully automatic segmentation, which requires no user input (Dou et al. 2024). While these approaches are powerful, they present challenges for use in games or animation, where the segmentation of objects is not always straightforward. It is not predetermined which parts of a model should be isolated as distinct segments, and this decision can vary based on artistic or functional requirements. Therefore, it is crucial that the user retains the ability to provide guidance, enabling the segmentation tool to extract precisely the parts needed for the intended application.

Two early and influential papers laid the foundation for segmentation in Gaussian splatting. SAGA (Cen et al. 2025) introduced a method to transfer segmentation from 2D images to 3D splats, allowing parts of a scene to be separated based on image data. Gaussian Grouping (Ye et al. 2024), in contrast, focused on assigning identity information to splats to group them into segments for later processing.

Numerous valuable papers have built upon the foundations established by SAGA and Gaussian Grouping. In particular, SAGA introduced the concept of using image masks as user input for segmentation. Each of these papers introduces useful advancements, and any one of them could have been selected as the basis for the continuation of this thesis (Lan et al. 2023; Lyu et al. 2025; Silva et al. 2024; Joseph, Amrutur, and Bhatnagar 2024). However,

one paper stood out as particularly well-suited to the objectives of this work: SAGD (Hu et al. 2025). This paper offers the added benefit of Gaussian Decomposition, which addresses a key challenge in segmentation. Specifically, at the boundary of a segment, where splats may overlap or extend beyond the desired cutoff, SAGD splits these splats into two. This results in a smoother and more precise boundary, eliminating the issues of missing or protruding splats. As discussed earlier, this capability is particularly relevant for the use case of this thesis, where cutting away sub-segments from a larger object often introduces chaotic and oversized splats along the boundary.

4.3 SAM

As described in the previous section, many of the papers relevant to this work, including SAGD, utilize image masks as user input. In most cases, these masks are generated by the Segment Anything Model (SAM) (Kirillov et al. 2023). SAM allows the user to provide positive and negative point prompts on the image, which guide the mask generation process. Additionally, bounding boxes can be used to further refine the selection. The result is a segmentation mask that includes all positive points, excludes negative points, and selects a coherent and visually meaningful region (see Figure 17).

Figure 17: Illustration of SAM segmentation with point prompts. From left to right: the original LEGO bulldozer; SAM segmentation with a single positive point on the rear wheel, selecting the entire bulldozer; and SAM segmentation after adding a negative point on another part, isolating only the wheel.

SAM achieves this by processing the input image through a vision transformer backbone, which encodes it into a high-dimensional feature representation. The user-provided prompts are embedded and integrated with these image features via a prompt encoder. A mask decoder then combines these elements to produce the final segmentation mask. This architecture allows SAM to leverage both spatial relationships and contextual information from the image, resulting in precise and adaptive segmentation that respects the user's input constraints.

4.4 SAGD

SAGD (Segment Anything via Gaussian Decomposition) closely aligns with the goals of this thesis. It not only segments Gaussian Splats based on user input, typically in the form of a SAM-generated mask, but also smoothens the boundary by splitting splats that overlap the segmentation edge.

The core process of SAGD begins with the user providing 2D prompts on one view of the scene. These prompts are mapped to their corresponding 3D

Gaussians, which are then reprojected into multiple other views to identify the same splats from different perspectives. For each view, SAM is applied to generate a segmentation mask that reflects the user's intent. Each Gaussian is then labeled based on a voting mechanism: if its projections consistently fall inside the masks across multiple views, it is included in the segment. To further improve the segmentation, SAGD employs Gaussian Decomposition at the segment boundary. This process splits overlapping splats into two parts—one included in the segment and the other excluded—producing a cleaner and more precise segmentation boundary. The final result is a segmentation of the Gaussian Splat that is both user-guided and geometrically refined.

Fortunately, the authors of SAGD provide an open-source implementation of their method. However, this implementation does not include its own test data. As a result, it was necessary to identify a suitable pre-trained scene for evaluation. Unlike the previous part of this thesis, this time the implementation required not only a `.ply` file containing the splats but also detailed information about the camera positions used to generate the Gaussian Splat. Conveniently, several scenes from the original Gaussian Splatting paper were available for download, complete with the required camera information. Unfortunately, the scene with the LEGO bulldozer was not among them. The criteria for selecting a scene remained consistent: it needed to be simple, clear, and easy to segment. Among the available options, the truck scene emerged as the most suitable choice, as illustrated in Figure 18. This scene was also used as an example in the SAGD paper itself.

Figure 18: The truck scene, a pre-trained Gaussian Splat model used for segmentation testing. Rendered in Unity.

With a suitable scene now available, the SAGD implementation could be tested. The provided code is written in Python, specifically within a Jupyter notebook environment. To generate the image corresponding to each camera position, the implementation renders the Gaussian Splat from the perspective of each camera. This process is computationally intensive, as it must render for every camera position. Fortunately, the downloaded scene included the original images captured from these camera positions during the training of the splat data. Thus, in the notebook, an adaptation was made to load these pre-existing images in place of rendering new views, significantly reducing runtime. For validation, an image mask was recreated on the same frame as the example in the original SAGD paper. Initial testing focused on the projection step, examining how the system mapped the selected points into other views. This was achieved by displaying the masks generated for each view, as illustrated in Figure 19

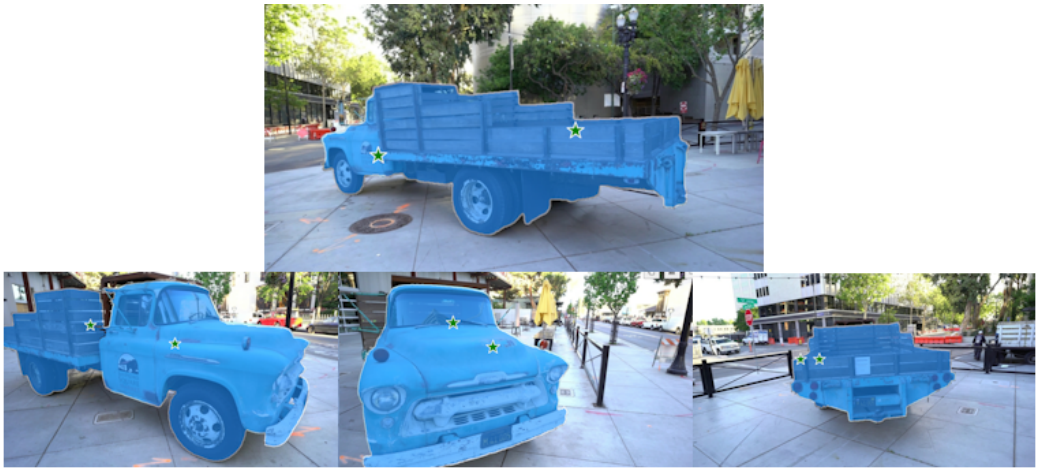

Figure 19: Testing the SAGD implementation. Top: input image with two positive points provided to SAM and the corresponding segmentation mask. Bottom: three different camera viewpoints, each displaying the projected prompt-points and the mask generated by SAGD for that camera position, all correctly reflecting the intended segmentation.

With the recreated example from the paper showing promising results, additional input points were tested on the same frame. Specifically, the next test focused on evaluating how SAGD handles segmenting the wooden platform of the truck. However, this test revealed multiple issues, as illustrated in Figure 20.

Figure 20: Testing the SAGD implementation. Top Left: input image with two positive points provided to SAM and the corresponding segmentation mask. The other three: three different camera viewpoints, each displaying the projected prompt-points and the mask generated by SAGD for that camera position. Notable issues with the segmentation are visible.

First, it should be noted that the frames shown in Figure 20 have been selectively chosen to illustrate the limitations observed. While a substantial number of frames yielded correct segmentations, there were several additional frames beyond the three presented that displayed issues. These issues can be categorized into two core limitations of the implementation:

- Convergence: In some cases, input points that are far apart in the original frame provide strong information for segmentation. However, under different viewing angles, these points may appear much closer together, reducing the specificity of the segmentation prompt. This convergence makes it more difficult for SAM to determine the intended segment, as observed in the top-right frame of Figure 20. While this issue can be somewhat mitigated by adding additional positive points,

it is possible for all available points to fall on the same plane in 3D space. In such cases, certain viewing angles may still cause clumping of the projected points, resulting in reduced prompt effectiveness.

- Occlusion: In the original frame, the input points are clearly located on a specific, unobstructed part of the object. However, when these points are projected into other views, changes in perspective or camera angle can cause parts of the object to become occluded by other elements of the scene. This means the projected points may land on areas of the image that visually correspond to different objects or regions, which SAM interprets as valid parts of the segment. As a result, the mask generated in these frames includes unintended areas. This issue is evident in the bottom two frames of Figure 20. Notably, in this scenario, adding additional positive or negative points cannot resolve the issue. Regardless of the other prompts, SAM will consistently attempt to generate a mask that includes the incorrectly placed point due to its interpretation of the user's intention.

Before exploring alternative methods of generating masks from other camera standpoints, it is both useful and logical to continue testing the remaining key components of SAGD: the voting system and the Gaussian Decomposition process. To evaluate these, the example will be revisited that was provided in the original SAGD paper—the segment that consistently produced accurate results. Figure 21 illustrates the output generated by the voting system alone, while Figure 22 shows the same output after Gaussian Decomposition has been applied.

Figure 21: Result of applying SAGD on the example from the paper, without using Gaussian Decomposition. The segmentation is relatively clean, except for a missing portion at the front of the object.

Figure 22: Result of applying SAGD on the example from the paper, with Gaussian Decomposition applied. The splats extending beyond the segment boundary at the bottom have been effectively removed. However, some additional minor errors have been introduced, such as a hole appearing in the black square behind the rear wheel.

The absence of the front of the truck in the segmentation is an undesirable outcome. While this issue is correctable, it will be addressed after the challenges identified earlier are resolved. Regarding the Gaussian Decomposition process, it performs its intended role of cleaning up boundary splats. The errors introduced are likely due to a combination of factors: imperfect initial segmentation data and the need for further fine-tuning of the decomposition frequency (as decomposition is not applied to every frame). However, refining this frequency is not a primary focus of this thesis.

4.5 SAM2

Throughout this section, it has been emphasized that an ideal implementation would accept a single SAM mask as input and automatically generate corresponding masks from different camera angles. This approach strikes an optimal balance between user control and automation: it enables the user to define a precise sub-object, while eliminating the need to manually create multiple masks from different perspectives—a process that would be nearly as cumbersome as the original method of extracting segments.

To reiterate, the ideal algorithm would transform a single SAM mask into multiple masks from different angles, without encountering the same issues as the original SAGD projection approach. This approach would enable the user to define just one mask while allowing SAGD to propagate it into a complete segmentation across the Gaussian Splat. This process would ultimately result in a perfectly smooth separation of the subsegment from the rest of the scene.
During the exploration of methods to generate these additional 2D masks, a promising development was discovered through the SAM documentation: SAM2 (Ravi et al. 2024). This extension of SAM introduces video segmentation capabilities. Its input structure is similar to that of image segmentation, with the key difference being that prompts can be placed on specific frames,

guiding segmentation in those particular frames. SAM2 can then generalize these prompts to produce a segmentation mask for every frame in the video (see Figure 23).

Figure 23: Result of SAM2 on a sample video provided by the SAM2 repository. Top-left: input frame with a single positive prompt point and the generated segmentation mask. The remaining images show the masks generated by SAM2 on other frames of the video, illustrating the model's ability to propagate the segmentation across frames. Note that in the top-right frame, the pants are almost entirely obstructed, yet the model still returns the correct mask—highlighting a clear improvement over the occlusion issues encountered in the original SAGD algorithm.

SAM2 builds upon SAM's image segmentation capabilities by extending them to video sequences. The user can provide segmentation prompts—such as points or bounding boxes—on any frame of the video. What sets SAM2 apart is its use of a memory mechanism that tracks and refines segmentation across frames. As the video progresses, SAM2 stores information from previous frames, including embeddings of both the frame features and the provided prompts. When processing a new frame, SAM2 retrieves this memory and

integrates it with the current frame's features to predict the segmentation mask. This enables SAM2 to maintain segmentation consistency even in the presence of occlusion, changes in object appearance, or movement.

To leverage the capabilities of SAM2, it is necessary to provide a continuous sequence of frames as input. One theoretical approach to generate such a sequence involves defining a continuous camera path around the object, with the camera oriented toward the object, rendering each frame to a file, and recording the corresponding world position of the camera in a separate file. However, a more straightforward approach is often available: the images used to train the Gaussian Splat model are frequently captured as a continuous video sequence, rather than as a collection of images from random camera positions. When access to these training images is available, as is the case for the truck scene used in this project, the task of creating a continuous sequence of frames is effectively simplified. By using these ordered images directly as input for SAM2, the system can treat them as a continuous video sequence without additional processing. This approach was chosen as it provided a practical and efficient means to integrate SAM2, allowing the focus to remain on addressing the core issues of SAGD without introducing the additional overhead of generating synthetic sequences.

To verify that the images form a continuous sequence, and to potentially aid in debugging, the separate images can be merged into a video by executing the following command in a Bash environment:

```
ffmpeg -framerate 30 -i %06d.jpg -c:v mpeg4 -q:v 5 output.mp4
```

The most straightforward approach to prompting a SAM2 video is to begin with the first frame and incrementally add positive and negative points to that frame until the generated mask closely aligns with the intended segment. However, SAM2 is not without its limitations. Due to its more sophisticated architecture compared to standard SAM, a greater number of points may

be required to obtain a satisfactory mask for the first frame alone (see Figure 24). Furthermore, while the masks generated for most frames may be acceptable, certain frames can still produce entirely inaccurate segmentations (see Figure 25). Unlike the projection-related issues encountered in SAGD, these problems can generally be resolved by adding more input points—either to the initial frame or directly to the problematic frames.

Figure 24: Progress of prompting the first frame in SAM2 to segment the wooden platform. Top-left: initial mask generated with two positive points. Subsequent images illustrate the addition of negative points, progressively refining the mask. The final frame (bottom-right) shows a mask that, while not perfect, achieves a satisfactory result across the video.

Figure 25: Segmentation of the entire truck using SAM2. Left: initial mask generated with only two positive points. Middle: a frame where the front of the truck is excluded from the mask due to its absence in some frames. Right: adding a single additional positive point to the problematic frame effectively resolves the issue.

SAM2's advantage lies in its ability to maintain temporal consistency and integrate contextual information from both the prompts and the sequence of frames. This enables it to produce reliable masks across the video with minimal additional input, contrasting with projection-based approaches where prompt points can easily shift to unrelated areas. As a result, the number of supplementary points required per segment remains limited, keeping the process significantly more efficient and user-friendly than manual segmentation or per-frame projection methods.

To demonstrate the reliability of SAM2, Figures 26 and 27 replicate the same experiment conducted with the projection-based approach. Starting from the same mask (requiring slightly more prompt points), the same frames were inspected. While some degree of selective framing has been used to highlight the contrast more effectively, it is important to note that any frame yielding an incorrect mask can be readily corrected by adding additional prompts.

Figure 26: Replication of the experiment from Figure 19, using SAM2 instead of projection.

Figure 27: Replication of the experiment from Figure 20, using SAM2 instead of projection.

Having demonstrated the potential of SAM2 to produce consistent and efficient segmentation masks across multiple viewing angles, the next logical step was to integrate these results with the SAGD pipeline. Instead of relying on per-frame projection methods to generate masks, the output of SAM2 was

leveraged directly. To facilitate this integration and streamline debugging, the generated masks were serialized into a pickle file—a standard Python format for storing and retrieving data structures such as dictionaries. In this case, the pickle file contained a mapping from each frame index to its corresponding segmentation mask. The SAGD implementation was then modified to read this file and use the precomputed masks as input for its downstream processing stages.

To validate this approach, the Gaussian Splat of the wooden platform was extracted using the same prompt points as in the earlier example. The resulting segmentation is illustrated in Figure 28.

Figure 28: Resulting Gaussian Splat of the wooden platform, generated using SAM2-based masks as input for the SAGD pipeline. The extracted segment accurately captures the platform's structure while excluding surrounding elements, illustrating the reliability of SAM2-generated masks for segmentation tasks.

4.6 Hierarchy

While the separation of a single segment is advantageous, the intended use case of this thesis involves interactive applications where both the subsegment and its parent object must coexist within the scene. Therefore, it is necessary to implement an intuitive and efficient method for segmenting a subcomponent while retaining both the isolated segment and the remainder of the object as distinct, non-overlapping entities.

The process of first segmenting the wooden platform of the truck and then performing a separate segmentation of the truck without the platform is not only impractical, but also prone to errors. This approach introduces the risk of including certain splats in both segments, resulting in incorrect visual artifacts. A hierarchical segmentation method, which preserves a clear parent-child relationship between the segments, is thus essential for maintaining both visual fidelity and workflow efficiency.

The most straightforward approach to achieve hierarchical segmentation, while also providing maximal control to the user, is to employ a layered data structure. Specifically, this thesis defines the following recursive structure:

Code 11: Definition of the recursive data structure used to represent hierarchical segmentation prompts and their relationships in SAM2

```
@dataclass
class framePrompt:
  points: np.ndarray
  labels: np.ndarray
  frame_number: int

@dataclass
class LayeredMaskPrompt:
  frameprompts: list[framePrompt]
  subobjects: list['LayeredMaskPrompt']
```

This structure enables the user to define high-level parent objects, using SAM2 prompt points and labels, and to specify subordinate child objects by nesting additional LayeredMaskPrompt entries. This recursive arrangement naturally supports hierarchical parent-child relationships, allowing for flexible and precise segmentation.
To compute the segmentation masks for each frame, SAM2 processes this hierarchical prompt structure and generates frame-wise masks for each node in the hierarchy. These masks are stored in a similarly recursive format using a pickle file: each dictionary contains a masks key, holding the mask information for the respective frames, and a subobjects key containing a list of child dictionaries structured identically.

SAGD can directly process these recursive structures to generate the desired segmentation. To ensure that splats do not appear redundantly in both parent and child segments—and to enforce a clean hierarchical split—the algorithm operates recursively. First, for a given node in the hierarchy, it computes the mask over the 3D splats of the object using standard techniques. It then iterates over each subobject (child node), invoking the same recursive procedure on each. Each recursive call returns the complete set of splats associated with the child, regardless of whether those splats are unique to the child or shared with its descendants. The parent node aggregates these returned splat sets to identify which of its own splats have been claimed by its subobjects. After processing all children, the parent writes to the output PLY file only those splats that were not claimed by any child, ensuring exclusivity. Finally, the parent returns the complete set of splats assigned to itself—including those shared with its subobjects—allowing this information to propagate upward through the recursion.

While this approach offers substantial control to the user, it also places significant responsibility on them to ensure that the defined hierarchy is coherent

and meaningful. For instance, the user could define two entirely separate nodes with substantial overlap in their segmentation regions, resulting in duplicated splats in the overlapping areas. However, due to the intuitive interface and real-time visual feedback provided by SAM2, the risk of such errors can be mitigated without imposing a significant burden on the user.

To validate the correctness of the new hierarchical algorithm, a straightforward test configuration was devised: two prompts were defined—one describing the entire truck and one describing the wooden platform. The platform prompt was specified as a child node of the truck prompt, both utilizing the same prompt points as in the earlier experiments. The resulting segmentation is illustrated in Figure 29.

Figure 29: Hierarchical segmentation of the truck and wooden platform. Left: both segments are shown in their original positions, appearing as a single cohesive object. Right: the wooden platform has been translated slightly to visually demonstrate its separation from the truck and to highlight the precision of the segmentation. Notable artifacts include small areas where the wood remains connected to the truck—likely due to minor segmentation inaccuracies in SAM2—and holes in the truck where the platform was extracted, resulting from missing splat data in occluded regions and the platform's thin structure.

4.7 Multimask Automation

This section has explored the challenge of balancing user control with automation in the segmentation process. While the previously described hierarchical method offers extensive flexibility, it also places a considerable burden on the user to maintain logical consistency, particularly when defining complex hierarchies. In certain cases, it may be desirable to reduce user intervention, especially when the relationship between segments is sufficiently straightforward and can be inferred from the context of the object and scene. This motivates the search for a method that requires the user to annotate only a single segment, while automating the determination of hierarchical relationships, thereby simplifying the segmentation process without sacrificing accuracy.

Once again, the key enabler of this approach is SAM itself. During image segmentation, SAM provides an optional parameter, `multimask_output`, which, when set to true, instructs the model to generate multiple segmentation masks instead of a single one. Specifically, it returns three masks, each representing a different level of segmentation granularity (see Figure 30). These masks are hierarchically nested, with each smaller mask fully contained within the next larger one, effectively creating a parent-child relationship. Additionally, each mask is accompanied by a confidence score, providing an automated means to evaluate the quality of the segmentation proposals and to filter out suboptimal candidates.

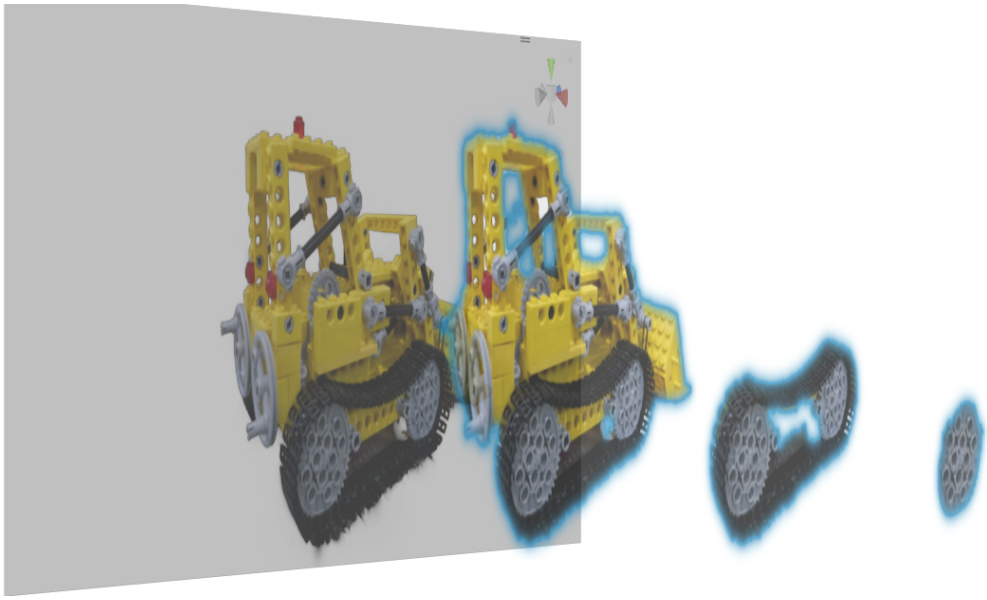

Figure 30: Demonstration of SAM's multilayer segmentation functionality on the LEGO bulldozer model. A single wheel was selected as the initial segment, resulting in three hierarchical masks: the smallest corresponds to the wheel itself, followed by the caterpillar tracks, and finally the entire bulldozer as the largest segment.

It is important to note that the `multimask_output` functionality is available only for image segmentation, not for video segmentation, which is the mode required for this project. However, video segmentation in SAM2 introduces an alternative feature that had not been utilized previously: in addition to providing prompts in the form of points or bounding boxes on a specific frame, the user can supply a full mask as input for that frame. This capability enables the following pipeline, as implemented in this thesis:
The user provides standard prompt points to define a target segment, from which the image segmentation algorithm generates three hierarchical masks. Applying a simple filter based on the quality scores of these masks, only the most confident masks are retained. This produces a local hierarchy of the segment. Each of these confident masks is then used as an input prompt

for video segmentation, resulting in corresponding masks for each frame in the video. The output data is then stored using the recursive data structure described in the previous section, with the hierarchical relationships already established by the multimask output. From this point onward, SAGD processes the input exactly as before, requiring no further modifications to the segmentation or rendering pipeline.

References

Cen, Jiazhong et al. (2025). *Segment Any 3D Gaussians*. arXiv: 2312.00860 [cs.CV]. URL: https://arxiv.org/abs/2312.00860.

Chao, Brian et al. (2025). *Textured Gaussians for Enhanced 3D Scene Appearance Modeling*. arXiv: 2411.18625 [cs.CV]. URL: https://arxiv.org/abs/2411.18625.

Dou, Bin et al. (2024). *Learning Segmented 3D Gaussians via Efficient Feature Unprojection for Zero-shot Neural Scene Segmentation*. arXiv: 2401.05925 [cs.CV]. URL: https://arxiv.org/abs/2401.05925.

Feng, Yutao et al. (2024). *Gaussian Splashing: Unified Particles for Versatile Motion Synthesis and Rendering*. arXiv: 2401.15318 [cs.GR]. URL: https://arxiv.org/abs/2401.15318.

Guédon, Antoine and Vincent Lepetit (2023). *SuGaR: Surface-Aligned Gaussian Splatting for Efficient 3D Mesh Reconstruction and High-Quality Mesh Rendering*. arXiv: 2311.12775 [cs.GR]. URL: https://arxiv.org/abs/2311.12775.

Guo, Jun et al. (2024). *Semantic Gaussians: Open-Vocabulary Scene Understanding with 3D Gaussian Splatting*. arXiv: 2403.15624 [cs.CV]. URL: https://arxiv.org/abs/2403.15624.

Hu, Xu et al. (2025). *SAGD: Boundary-Enhanced Segment Anything in 3D Gaussian via Gaussian Decomposition*. arXiv: 2401.17857 [cs.CV]. URL: https://arxiv.org/abs/2401.17857.

Ji, Shengxiang et al. (2024). *Segment Any 4D Gaussians*. arXiv: 2407.04504 [cs.CV]. URL: https://arxiv.org/abs/2407.04504.

Jiang, Yingwenqi et al. (2023). *GaussianShader: 3D Gaussian Splatting with Shading Functions for Reflective Surfaces*. arXiv: 2311.17977 [cs.CV]. URL: https://arxiv.org/abs/2311.17977.

Joseph, Joji, Bharadwaj Amrutur, and Shalabh Bhatnagar (2024). *Gradient-Driven 3D Segmentation and Affordance Transfer in Gaussian Splatting Using 2D Masks*. arXiv: 2409.11681 [cs.CV]. URL: https://arxiv.org/abs/2409.11681.

Kerbl, Bernhard et al. (2023). *3D Gaussian Splatting for Real-Time Radiance Field Rendering*. arXiv: 2308.04079 [cs.GR]. URL: https://arxiv.org/abs/2308.04079.

Kirillov, Alexander et al. (2023). "Segment Anything". In: *arXiv:2304.02643*.

Lan, Kun et al. (2023). *2D-Guided 3D Gaussian Segmentation*. arXiv: 2312.16047 [cs.CV]. URL: https://arxiv.org/abs/2312.16047.

Li, Yun-Jin et al. (2024). *SADG: Segment Any Dynamic Gaussian Without Object Trackers*. arXiv: 2411.19290 [cs.CV]. URL: https://arxiv.org/abs/2411.19290.

Liang, Siyun et al. (2024). *SuperGSeg: Open-Vocabulary 3D Segmentation with Structured Super-Gaussians*. arXiv: 2412.10231 [cs.CV]. URL: https://arxiv.org/abs/2412.10231.

Lyu, Weijie et al. (2025). *Gaga: Group Any Gaussians via 3D-aware Memory Bank*. arXiv: 2404.07977 [cs.CV]. URL: https://arxiv.org/abs/2404.07977.

Mildenhall, Ben et al. (2020). "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *ECCV*.

Navaneet, KL et al. (2024). *CompGS: Smaller and Faster Gaussian Splatting with Vector Quantization*. arXiv: 2311.18159 [cs.CV]. URL: https://arxiv.org/abs/2311.18159.

Nguyen, Thanh Tuan, Phuong Nguyen, and Frederic Bouchara (Aug. 2023). "Representing dynamic textures based on polarized gradient features". In: *Machine Vision and Applications* 34. DOI: 10.1007/s00138-023-01438-7.

Pranckevičius, Aras (2023). *UnityGaussianSplatting*. Accessed: 2025-02-16. URL: https://github.com/aras-p/UnityGaussianSplatting.

Ranjan, Darsh (2021). *python-plyfile*. URL: https://github.com/dranjan/python-plyfile.

Ravi, Nikhila et al. (2024). *SAM 2: Segment Anything in Images and Videos*. arXiv: 2408.00714 [cs.CV]. URL: https://arxiv.org/abs/2408.00714.

Rubloff, Michael (Dec. 2024). *Gaussian Splatting Unity Plugin gets VR Support*. Accessed: 2025-02-16. URL: https://radiancefields.com/gaussian-splatting-unity-plugin-gets-vr-support.

Sathyamoorthy, Dinesh and Radhakrishnan Palanikumar (Jan. 2006). "Linear and nonlinear approach for DEM smoothening". In: *Discrete Dynamics in Nature and Society* 2006. DOI: 10.1155/DDNS/2006/63245.

Saucier, Nathan (June 2024). *Gaussian Splatting: Rapid 3D with AI Tools*. Accessed: 2025-02-16. URL: https://www.leidenlearninginnovation.org/stories/gaussian-splatting-rapid-3d-with-ai-tools/.

Schaefer, S. and J. Warren (2004). "Dual marching cubes: primal contouring of dual grids". In: *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.* Pp. 70–76. DOI: 10.1109/PCCGA.2004.1348336.

Shi, Jin-Chuan et al. (2023). *Language Embedded 3D Gaussians for Open-Vocabulary Scene Understanding*. arXiv: 2311.18482 [cs.CV]. URL: https://arxiv.org/abs/2311.18482.

Silva, Myrna C. et al. (2024). *Contrastive Gaussian Clustering: Weakly Supervised 3D Scene Segmentation*. arXiv: 2404.12784 [cs.CV]. URL: https://arxiv.org/abs/2404.12784.

Smith, Thomas (2024). *GPUSorting*. URL: https://github.com/b0nes164/GPUSorting.

Ye, Mingqiao et al. (2024). *Gaussian Grouping: Segment and Edit Anything in 3D Scenes*. arXiv: 2312.00732 [cs.CV]. URL: https://arxiv.org/abs/2312.00732.